Lecture 15: OOP

CS 61A, Summer 2006

Instructor: Kevin Lin

Administrative stuff

- I am leaving Berkeley this Saturday. If you need to talk to me please contact me as soon as possible so that we can meet tomorrow.
- Reminder: Regrade deadline for hw1ab, hw2ab, proj1, mt1 is Monday. You must at least send us an e-mail by 11 AM. See yesterday's notes for more details.

One, two, three!

- proj1 paper copies should be handed back in discussion.
- proj2 test cases: we're not going to care about them that much. Don't waste gallons of ink printing out screenshots.
- proj3 posted. It will be done in pairs. ONLY ONE PERSON NEEDS TO SUBMIT PER GROUP. There are two parts, the first due in one week and the second due in two weeks. Photocopies of project specification will be handed out.

OOP

- Photocopies of OOP "above the line" notes will be handed out (contains OOP reference manual as well).
- "Above the line" refers to the actual use of the language; we ignore how it's implemented. "Below the line" refers to the implementation or possible implementations of the language.
- To do hw4b and proj3, you mostly need to know "above the line" stuff.

OOP Review

```
(ask object message arg1 arg2 ...)
```

Ask gets a <u>method</u> from an <u>object</u> corresponding to the <u>message</u>. If the object has such a method, invoke it with the given arguments; otherwise there is an error.

```
(instantiate class arg1 arg2 ...)
```

Instantiate creates a new <u>instance</u> of a <u>class</u>. The extra arguments are <u>instantiation variables</u>. Such instances are implemented (in our implementation of OOP) as procedures.

OOP Review

```
(define-class (class-name arg1 arg2 ...)
  clause1
  clause2
  ...)
```

Defines a <u>class</u> with <u>instantiation variables</u> arg1, arg2, ...

Clauses in class definitions

```
(method (message arg1 arg2 ...)
  body ...)
```

Gives the class a method corresponding to a message, with the given arguments and body.

```
(instance-vars (var1 val1)
               (var2 val2) ...)
```

Sets up local state variables val1, val2, etc. Each instance gets its own copy of each of these. These are visible inside the bodies of the methods and the initialization code within the same class definition. They are not visible outside, except via (ask some-object 'instance-var-name).

Clauses in class definitions

```
(class-vars (var1 val1)
            (var2 val2) ...)
```

Also sets up local state variables var1, var2, etc. Except now, all instances of a given class SHARE one copy of each of the variables. Also accessible via (ask some-object 'class-var-name)

```
(parent (parent1 args ...)
        (parent2 args ...))
```

If a message is not recognized, we will pass the message on the parent1. If parent1 does not recognize it, then we'll try parent2, etc.

Clauses in class definitions

```
(default-method body ...)
```

Specifies the code that an object should execute If it receives an unrecognized message.

```
(initialize body ...)
```

The body is executeed whenever an instance of this class is created.

OOP

The most complicated thing about OOP is inheritance. Consider these complicated classes:

```
(define-class (class1)
  (method (f) 1)
  (method (b) (ask self 'f)))
(define-class (class2)
  (parent (class1))
  (method (c) 3)
  (method (e) (ask self 'f)))
(define-class (class3)
  (parent (class2))
  (method (e) (+ 1 (usual 'e)))
  (method (f) 4))
```

OOP

```
(define-class (class1)
  (method (f) 1)
  (method (b) (ask self 'f)))
(define-class (class2)
  (parent (class1))
  (method (c) 3)
  (method (e) (ask self 'f)))
(define-class (class3)
  (parent (class2))
  (method (e) (+ 1 (usual 'e)))
  (method (f) 4))
(ask (instantiate class3) 'c) => ?
(ask (instantiate class1) 'b) => ?
(ask (instantiate class2) 'e) => ?
(ask (instantiate class3) 'e) => ?
(ask (instantiate class3) 'b) => ?
```

Implementing a simple put/get table with OOP & set!

NOTE: This is NOT the put/get used in data-directed programming. I'm only using this as an example. Do NOT use this on questions that have to do with data-directed programming.

```
> (define table1 (instantiate table))
> (ask table1 'put 'x 1)
> (ask table1 'get 'x)
1
> (ask table1 'get 'y)
#f

etc.
```

Implementing a simple put/get table with OOP & set!

In this version, the put method only takes two arguments.

How can you extend this to create a new version of tables in which the put method works with three arguments?

13

OOP paradigm

The OOP paradigm has become very popular in recent years. In Java, almost everything you deal with is an object. The same thing is true in many other modern programming languages.

It is a very natural and easy way to state many types of problems.

There are many styles and paradigms of programming. Some are better suited for certain tasks than others.

14

Set! (pronounced “set bang”)

Set! is a special form.

It is an example of what we call mutation; it changes bindings of variables without creating a brand new binding.

```
(define x 1)
(define (f y) (set! x y))
(f 2)
```

After this call, x evaluates to 2

15

Implementing OOP and the environment model

Moving on …

STk and chalkboard …

16

Implementing OOP and the environment model

```
(define (count) ;does this work as desired?
  (let ((x 1))
    (set! x (+ x 1))
    x))

(define count  ;this works!
  (let ((x 1))
    (lambda ()
      (set! x (+ x 1))
      x)))

(define (make-count)  ;this dispenses count procs
  (let ((x 1))
    (lambda ()
      (set! x (+ x 1))
      x)))
```

17